Projet d'algorithmique II

Daniel Zhu

Le 27 janvier 2019

Préambule

Dans la continuité du projet d'algorithmique précédent, on s'intéresse ici à la création d'un algorithme capable d'explorer la totalité d'un labyrinthe sans dépasser un certain nombre de mouvements – ce qui revient à une mort par épuisement.

On ne peut pas se baser sur le même algorithmique utilisé dans le projet précédent puisque celui-ci cherchait à rejoindre un endroit précis du donjon, à l'aide d'une boussole et d'un indicateur métrique que l'on n'a de toute façon pas accès ici.

Mais l'utilisation de structures de données telles que des arbres – ici quaternaires – et de listes chaînées permettront de proposer une solution au problème, comme nous allons le voir dans ce rapport.

1 Stratégie

Le labyrinthe est représentable par un arbre dont chaque noeud correspond à un endroit du niveau et chaque sous-enfant, lorsqu'il existe, la position adjacente au noeud parent.

On peut donc explorer le labyrinthe tout entier en parcourant la totalité de l'arbre, selon un ordre précis – par exemple, Nord, Est, Sud puis Ouest.

Cette stratégie a toutefois l'inconvénient de baser la condition d'arrêt d'exploration au retour de Thésée à sa position de départ. En effet, on explore chaque branche jusqu'à la feuille mais on remonte toujours vers la

racine, ce qui peut augmenter le risque d'épuisement, *a fortiori* lorsque le labyrinthe est vaste.

Il doit sûrement exister des moyens pour éviter ce risque mais on ne s'y est pas préoccupé, estimant qu'un autre problème plus sérieux était à prendre en compte : les boucles.

En effet, lorsque Thésée s'engage dans une portion du labyrinthe qui réalise une boucle, il revient nécessairement à des noeuds déjà visités et enregistrés dans l'arbre. Mais ce dernier se contente de dupliquer ces noeuds, ce qui complexifie considérablement le parcours et donc l'exploration du labyrinthe.

Une solution possible à ce problème consiste à utiliser un fil d'Ariane que l'on a dû implémenter à l'aide de listes chaînées.

Ce fil contient dans l'ordre chronologiquement décroissant les mouvements effectués par Thésée (du plus récent au plus ancien).

L'idée consiste à faire une prédiction de boucle : si, en ajoutant le prochain mouvement que l'on s'apprête à faire au fil d'Ariane, on détecte une boucle, alors on ordonne à Thésée de faire demi-tour : c'est ce qu'on appellera ici la « procédure antiboucle ».

Mais ce n'est pas suffisant. En effet, en revenant au carrefour de la boucle, puisque Thésée a emprunté un chemin et pas l'autre, il voudra prendre l'autre chemin puisque le sous-noeud associé à celui-ci n'est pas encore rempli et qu'il a techniquement la possibilité de l'emprunter. On active alors la « procédure embuscade », l'idée qu'une autre boucle attend Thésée alors qu'il vient d'en éviter une (la même...)

Cette fois-ci, on ne se base plus sur le fil d'Ariane mais plutôt sur le noeud position P, là où se situe actuellement Thésée, qui s'apprête à faire le mouvement m. On explore récursivement chaque sous-noeud de P puis on génère pour chaque sous-branche i un fil d'Ariane f_i depuis la position actuelle de Thésée P_m jusqu'à la feuille de chaque sous-branche de P.

Puis on vérifie que la position absolue d'arrivée de chaque fil d'Ariane hypothétique $f_i+\{m\}$ ne coïncide pas avec la position absolue qu'atteindrait Thésée s'il effectuait le mouvement m. En clair, pour que m soit effectué, on doit avoir $\mathrm{position}(f_i+\{m\}) \neq \mathrm{position}(\{P_m\}+\{m\})$, où $\mathrm{position}(x)$ indique la position absolue de Thésée dans le labyrinthe suivant un chemin x.

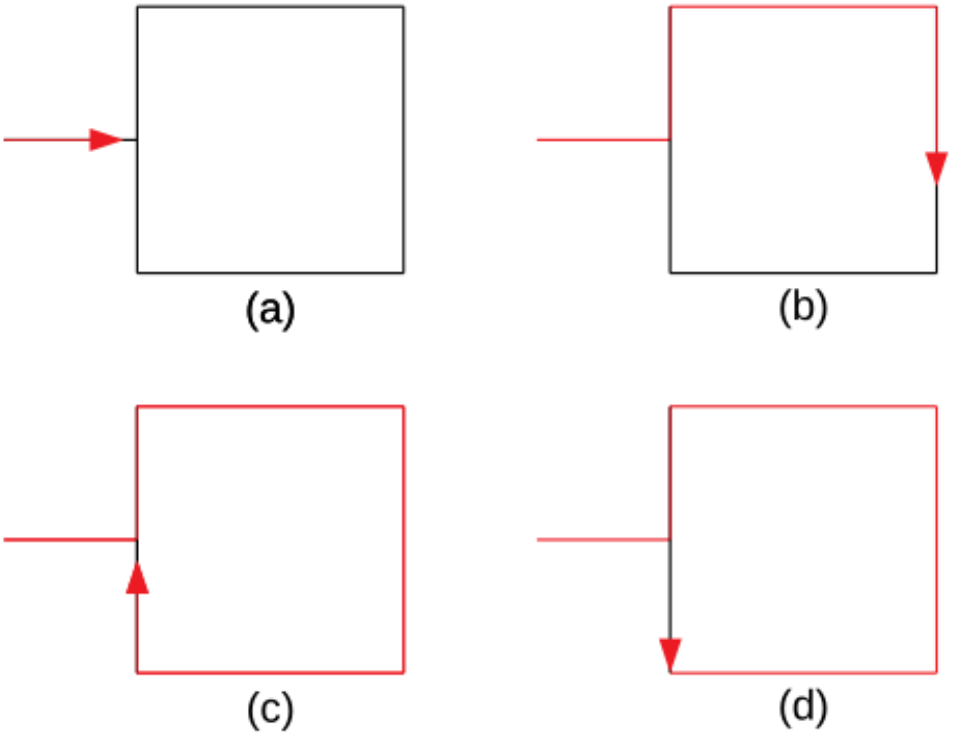

FIGURE 1 – Déroulement de la procédure antiboucle – (a) Thésée s'apprête à entrer dans une boucle ; (b) Il parcourt la boucle ; (c) Mais ne la finit pas et s'arrête juste devant le carrefour : en effet, s'il continuait, il atteindrait un noeud déjà visité ; (d) Il fait donc demi-tour.

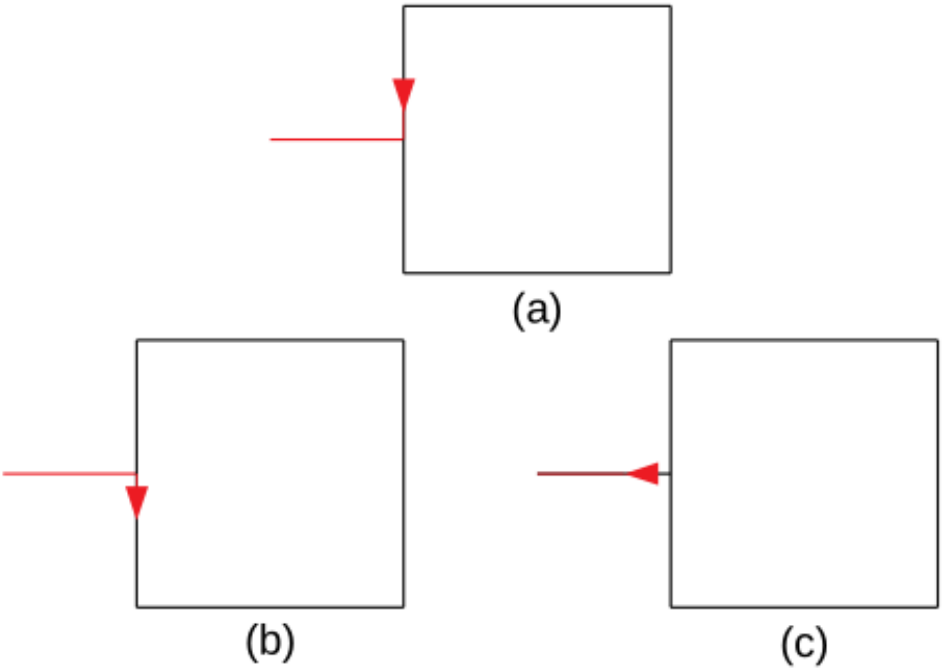

FIGURE 2 – Déroulement de la procédure embuscade – (a) Thésée revient au carrefour ; (b) Mais il n'a pas visité le chemin du bas d'après l'arbre (qui est en fait déjà visité via le chemin du haut), donc il prend ce chemin, or **c'est exactement ce qu'on veut éviter** ; (c) La procédure embuscade doit alors permettre à Thésée de savoir qu'il ne doit **pas** prendre le chemin du bas, et remonter dans le noeud parent, selon le protocole décrit plus haut.

2 Partie technique

2.1 Des fonctions et procédures

Toutes les fonctions et procédures ont été au maximum décrites en commentaire dans le code source.

Afin d'éviter d'imbriquer trop de structures conditionnelles ou de boucle, j'ai systématisé l'utilisation de fonctions et de procédures.

Ceci, d'abord pour améliorer la lisibilité du code (j'ai essayé de ne jamais dépasser 80 caractères par ligne, une « colonne » de code étant plus facile à lire qu'un « paysage » de code), et conséquemment, pour faciliter la maintenance (détection et correction des problèmes).

J'ai crée deux fonctions distinctes pour la procédure antiboucle et la procédure embuscade. La première est un simple détecteur de boucle comptant le nombre de mouvements cardinaux à partir d'un fil d'Ariane : elle s'arrête dès qu'une boucle est détectée et ne remonte donc pas jusqu'à l'origine. La deuxième remonte tout le fil d'Ariane puisque celui-ci est hypothétique et qu'il ne faut pas qu'elle détecte une boucle voisine non liée au prochain mouvement qu'on s'apprête à faire.

Des indications sont disponibles en mode débug, à condition de définir la constante globale `debugMode` sur `TRUE` au début du code source : cela affichera le fil d'Ariane, les mouvements où l'algorithme détecte une boucle, une embuscade, etc.

2.2 Des pointeurs

Il m'a semblé important de ne travailler qu'avec des pointeurs en « mode lecture uniquement » dans un souci de rigueur et de sûreté. Par exemple, pour créer le fil d'Ariane :

```
string const thread = malloc( sizeof(struct link) );
```

Comme `string` est en fait un `struct link *`, le mot-clé `const` ne s'applique qu'au pointeur. Ainsi, les membres de `thread` sont modifiables mais pas son adresse.

La généralisation du mot-clé `const` dans l'initialisation des pointeurs, des variables et des paramètres altère un peu le sens sémantique des constantes au sens algorithmique, mais je pense qu'il m'a aidé à éviter beaucoup d'erreurs, si bien qu'il me semble que les problèmes que j'ai eu durant le projet étaient plus d'ordres algorithmiques qu'à une mauvaise utilisation des pointeurs – du moins, je l'espère.

3 Conclusion

L'algorithme fonctionne pour tous les niveaux, du premier au huitième. Toutefois :

— étant conçu pour des labyrinthes simples, il a beaucoup plus de mal pour les labyrinthes avec des salles (surtout si elles sont grandes...) ;
— il n'est certainement pas le plus efficace puisque les stratégies déployées sont relativement simples et « naïves » ;
— des améliorations sont certainement possibles, notamment au niveau de la condition d'arrêt d'exploration ou même dans la génération du fil d'Ariane (parcours d'arbre pour trouver Thésée).

Néanmoins, je pense qu'il apporte une solution acceptable au problème demandé.